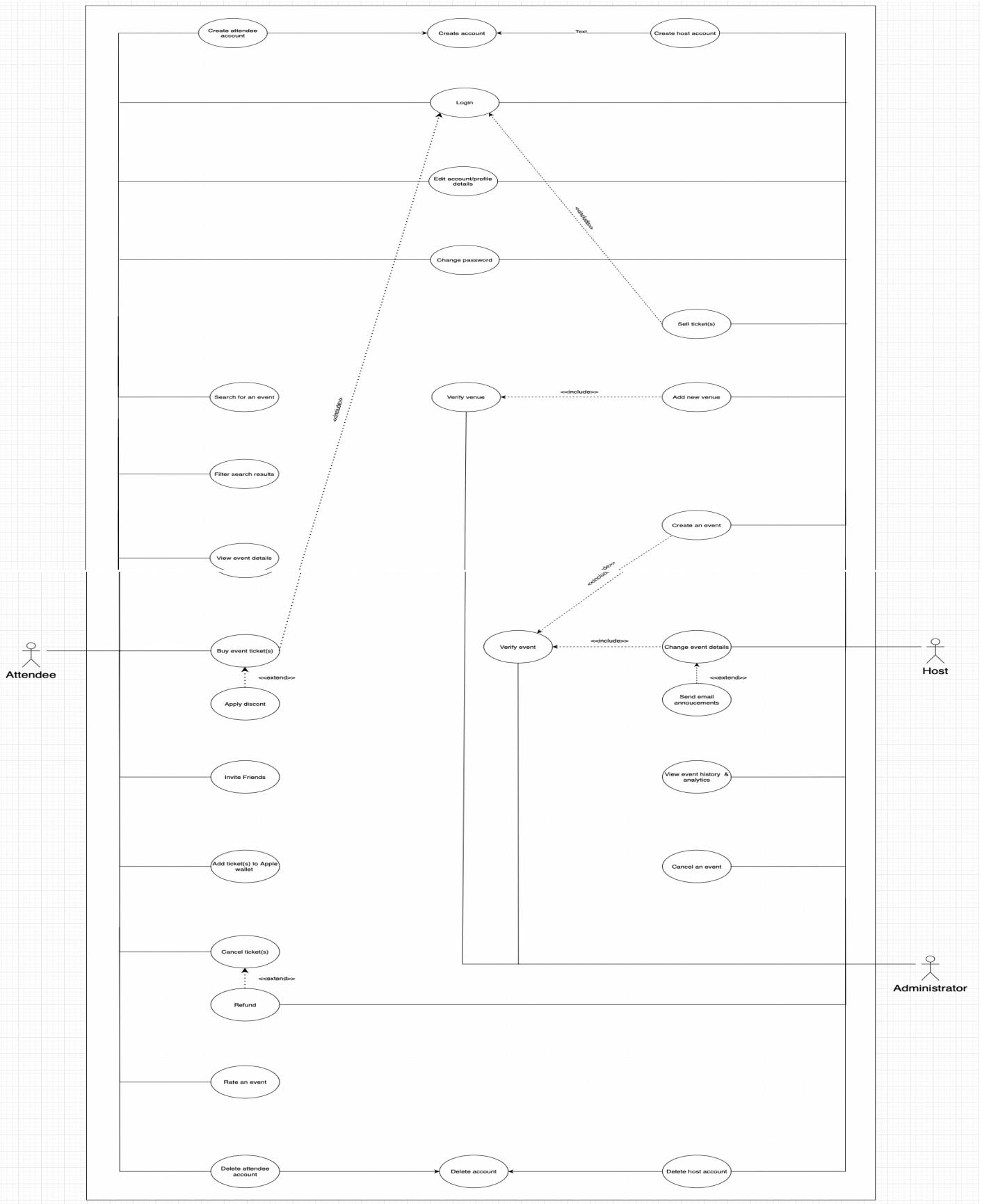
Group name: PASSIO (Adams, Yuki, Kaycee, Jared, Bryce)
Lab time: Friday 8-10am

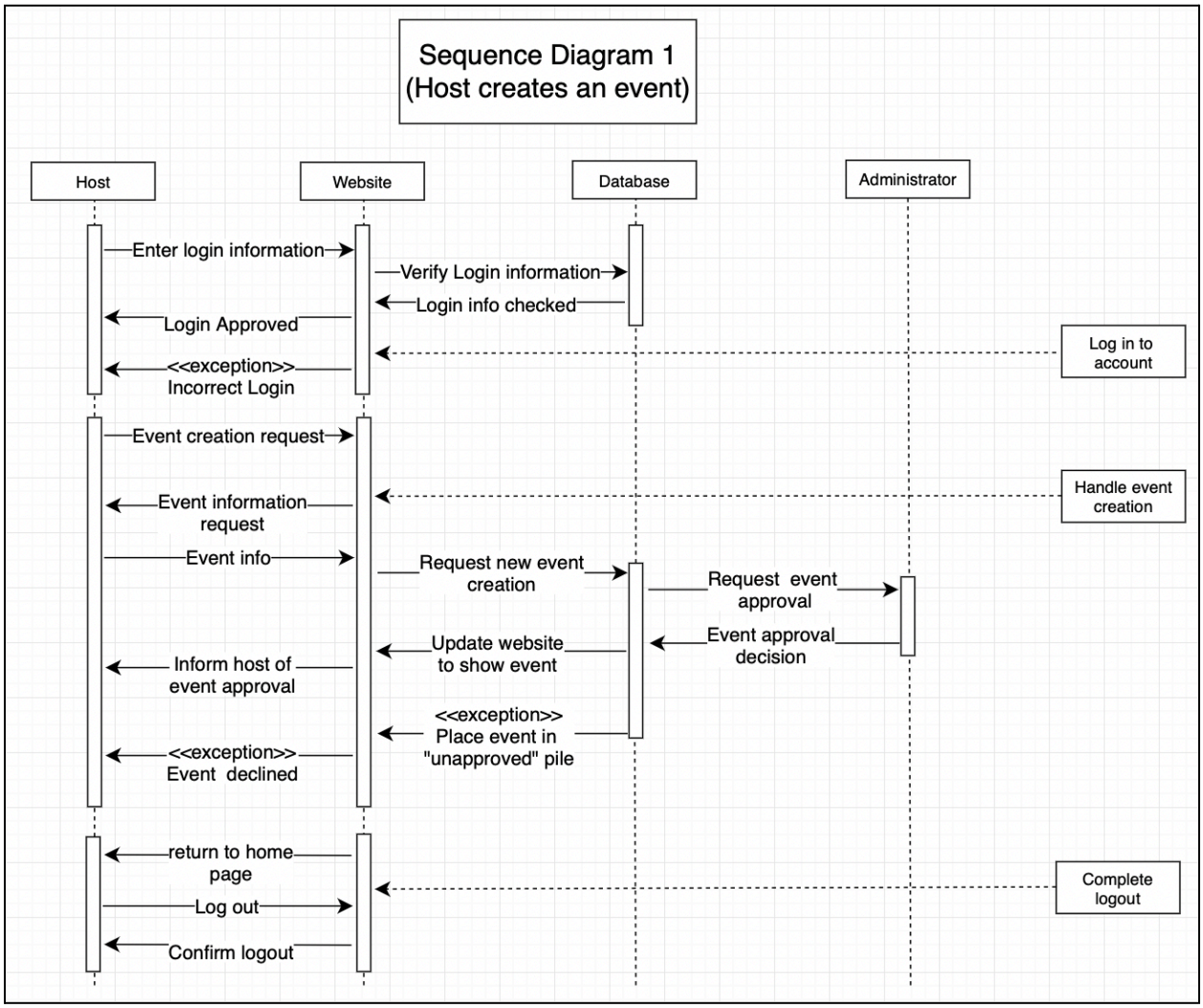
Milestone #3: Formal Analysis and Architecture (Due March 8th)

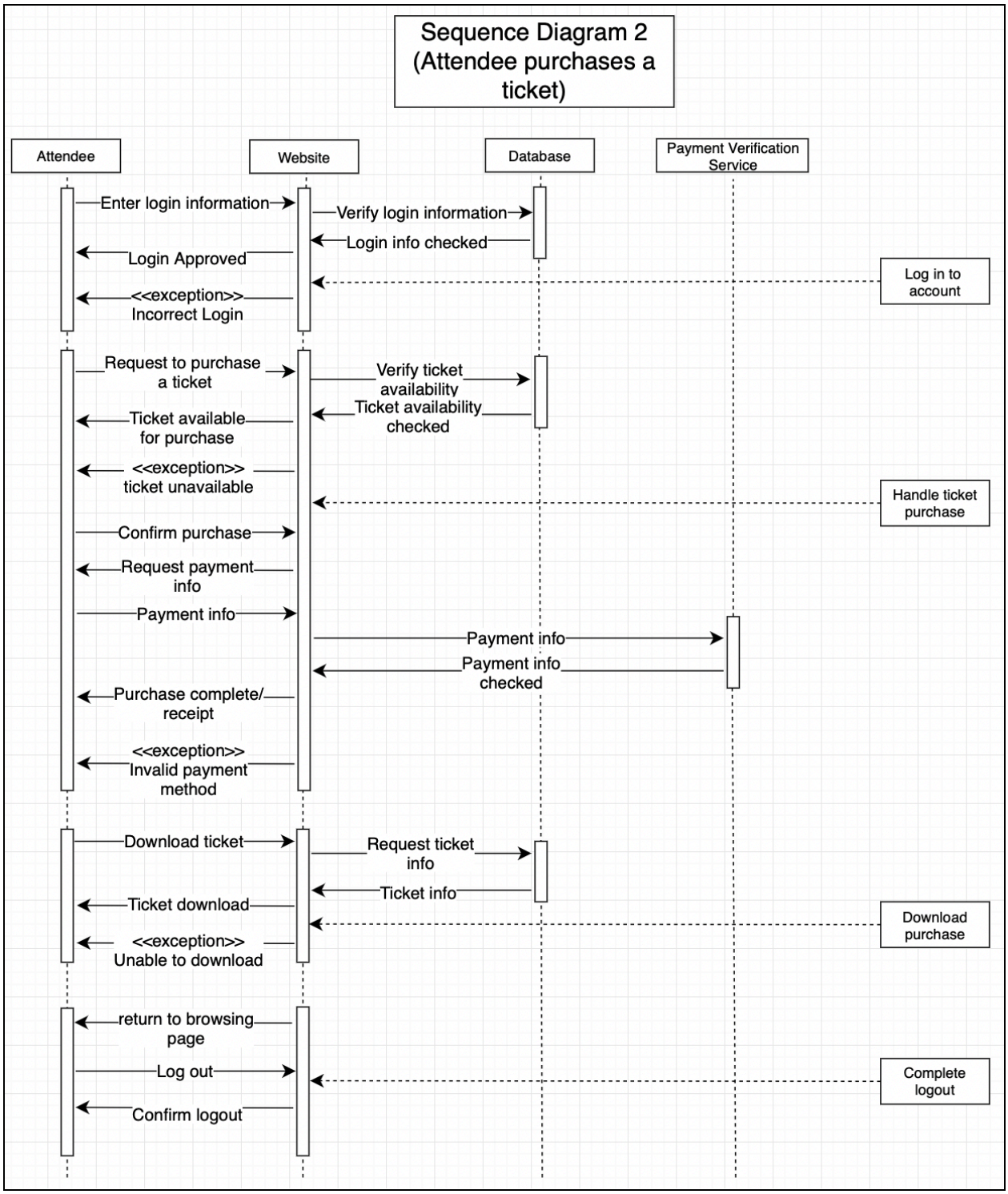
Link to our draw.io to see diagrams in more detail (if desired) →
https://drive.google.com/file/d/1ex35gwBd36yW-yKLhSZwxmrXEQ1WFWOb/view?usp=sharing

1. **Use Case Diagram (from draw.io)**
    a. Sorry about the formatting for the use case diagram. It was quite large, so we took a photo of the top and the bottom as it would be illegible if we took a photo of it all together. The rest of the diagrams are clear (scroll down to see).

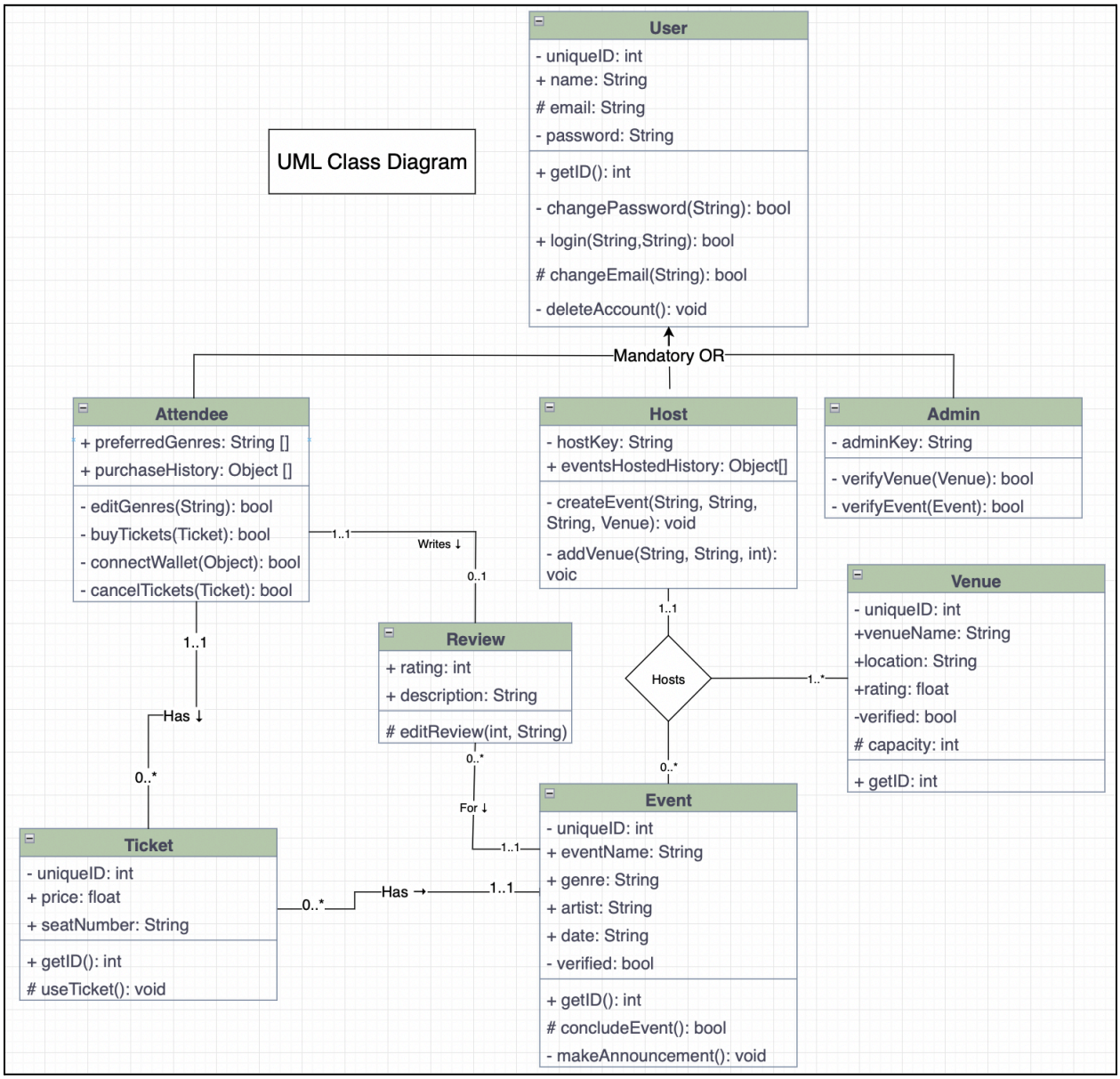

Create attendee account
Create account
Text
Create host account
Login
Edit account/profile details
Change password
Sell ticket(s)
Search for an event
Verify venue
<<include>>
Add new venue
Filter search results
Create an event
View event details
Buy event ticket(s)
Verify event
Change event details
Attendee
Host
<<extend>>
Apply discont
Send email annoucements
Invite Friends
View event history & analytics
Add ticket(s) to Apple wallet
Cancel an event
Cancel ticket(s)
Administrator
Refund
Rate an event
Delete attendee account
Delete account
Delete host account

2. Sequence diagrams x 2 (from draw.io)

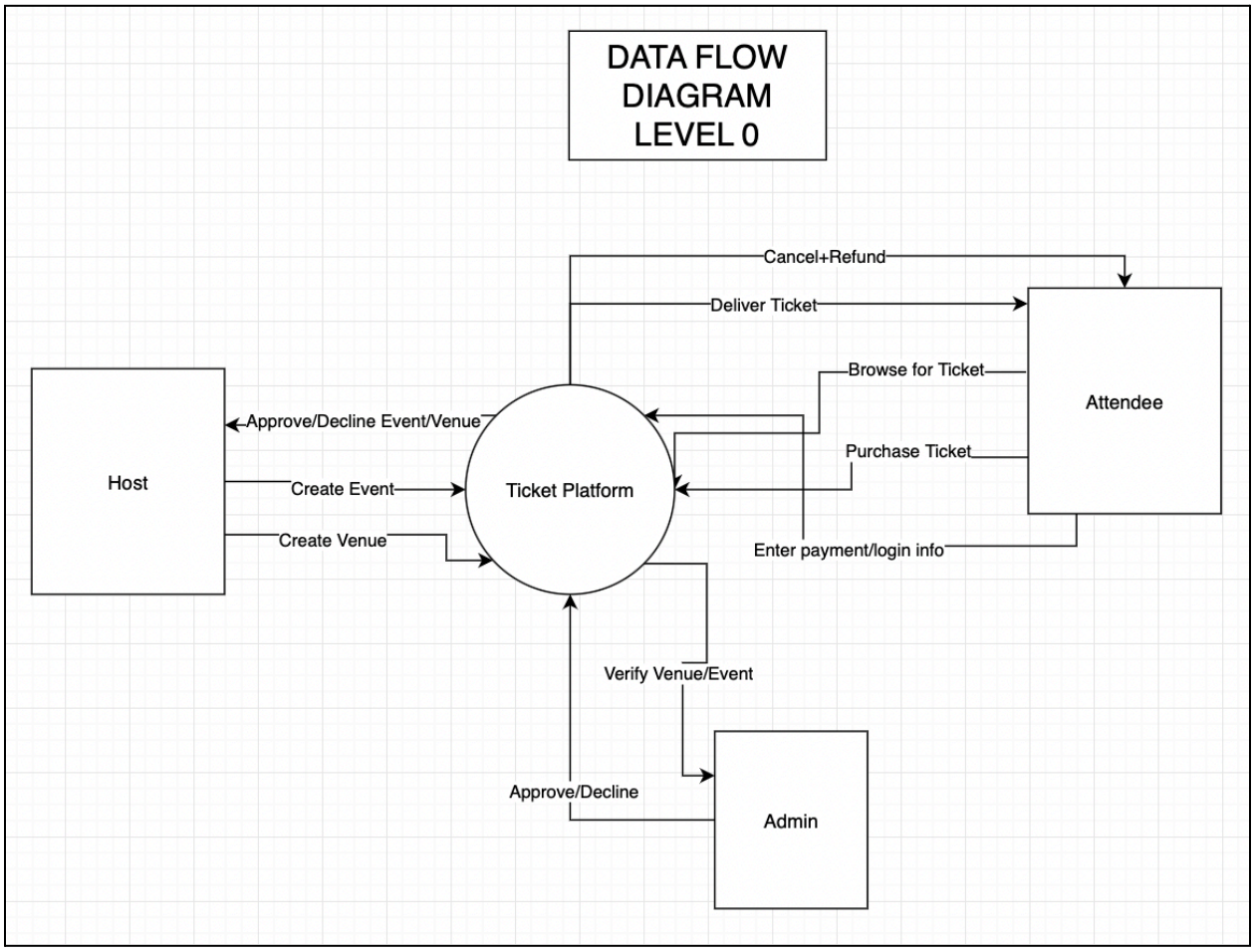



Sequence Diagram 2 (Attendee purchases a ticket)
Attendee
Website
Database
Payment Verification Service
Enter login information
Verify login information
Login info checked
Login Approved
<<exception>> Incorrect Login
Log in to account
Request to purchase a ticket
Verify ticket availability
Ticket availability checked
Ticket available for purchase
<<exception>> ticket unavailable
Handle ticket purchase
Confirm purchase
Request payment info
Payment info
Payment info
Payment info checked
Purchase complete/ receipt
<<exception>> Invalid payment method
Download ticket
Request ticket info
Ticket info
Ticket download
Download purchase
<<exception>> Unable to download
return to browsing page
Log out
Complete logout
Confirm logout

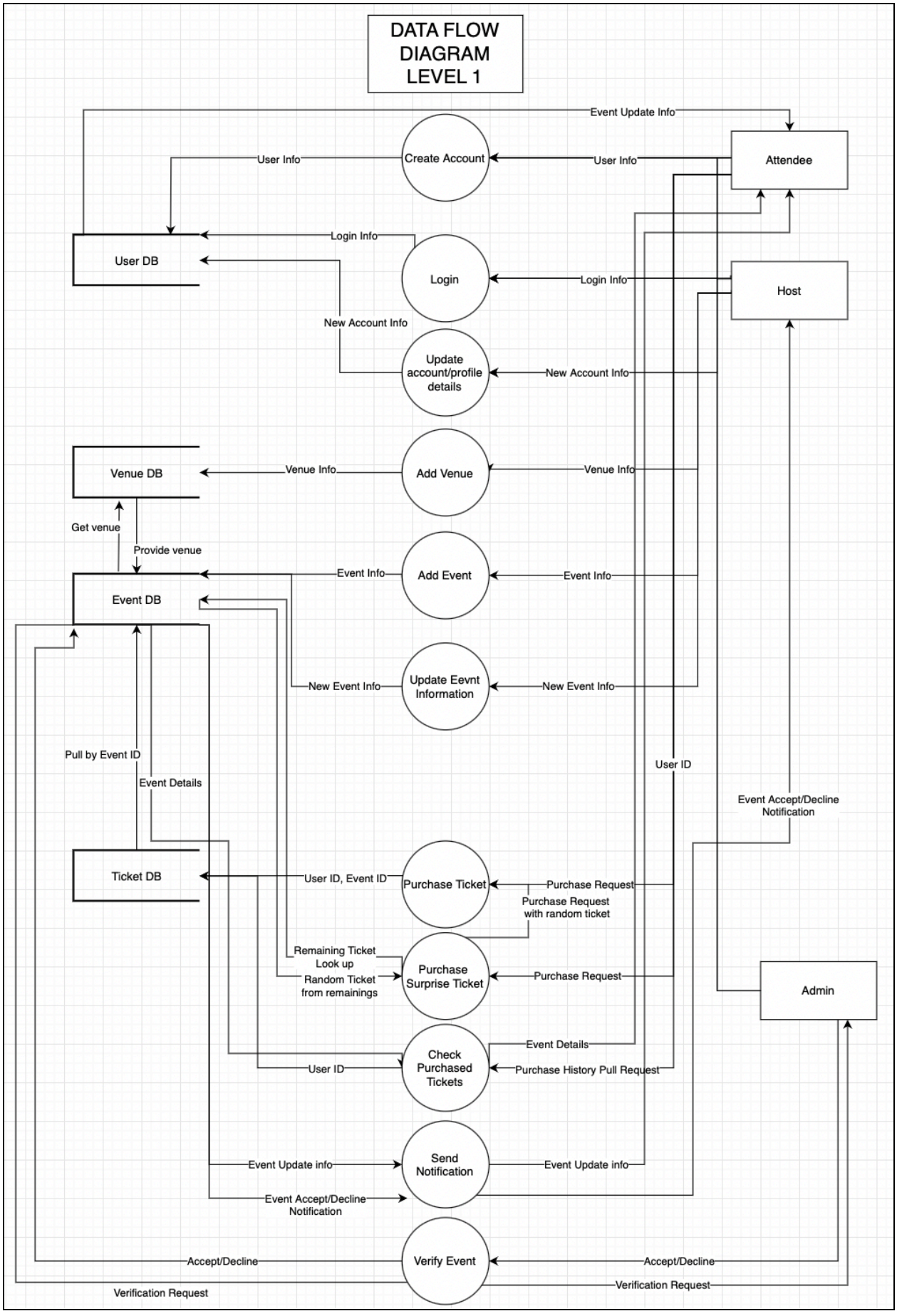
3. UML Class Diagram (from draw.io)



4. **Data Flow Diagram (Level 0) (from draw.io)**



5. **Data Flow Diagram (Level 1) (from draw.io)**


DATA FLOW DIAGRAM LEVEL 1
Event Update Info
Create Account
User Info
User Info
Attendee
User DB
Login Info
Login
Login Info
Host
New Account Info
Update account/profile details
New Account Info
Venue DB
Venue Info
Add Venue
Venue Info
Get venue
Provide venue
Event DB
Event Info
Add Event
Event Info
Update Eevnt Information
New Event Info
New Event Info
Pull by Event ID
Event Details
User ID
Event Accept/Decline Notification
Ticket DB
User ID, Event ID
Purchase Ticket
Purchase Request
Purchase Request with random ticket
Remaining Ticket Look up
Random Ticket from remainings
Purchase Surprise Ticket
Purchase Request
Admin
Check Purchased Tickets
Event Details
User ID
Purchase History Pull Request
Send Notification
Event Update info
Event Update info
Event Accept/Decline Notification
Verify Event
Accept/Decline
Accept/Decline
Verification Request
Verification Request

6. **Test Plan**
    a. Starting from a general overview, we aim to implement both static and dynamic testing procedures within our code design. We wanted to have the items discussed in this document in mind while we write our code and then during the merging of pull requests, we will go through the code and assess for the features we discuss below. This way we hoped we would have easily maintainable code, that is easy to debug and has good readability, testability, and configurability across our chosen platform.
    b. STATIC TESTING: For our static procedures, we want to focus on having functional, maintainable code. For this we plan to utilize the summarized inspection checklist for common errors, provided to us on slides 35/36 in Lecture 1 of week 5 for COSC 310. This is expanded on below with our main focal points emphasized:
        i. Data Faults:
            1. For each program block, we aim to have code blocks with the least amount of syntax error. For this purpose, each developer is responsible to check for the syntax error for the code block he wrote. More specifically, we need to make sure that all variables are declared correctly and local variables do not overwrite global variables. During the merging of pull requests, we check through the code quickly to look for the syntax error.
        ii. Control Faults:
            1. We aim to have at least one test method for each method to ensure that the logic built in the method is correct. This test needs to run at every system test and total code coverage needs to be above 80%. The remaining 20% is for exception handling code which is costly to intentionally occur but poorly benefits us.
        iii. Exception Management Faults:
            1. To make sure all possible exceptions are handled, we need to have mainly two approaches. One is using try-catch which is mostly used to deal with the error triggered by unknown input such as I/O. This should be designed to bring the flow back to the normal flow even if the system faces unexpected input. The other possible error is logic error. Since we aim to have the least amount of error, instead of handling with try-catch, we fix the code to eliminate the potential of logic error as soon as we find it.
        iv. Input/Output Faults:
            1. Since many inputs are unknown, they are most likely to trigger errors. As previously stated in the Exception Management Faults section, We handle the possible exceptions by using try-catch as much as possible. Commonly, we need to pay attention to the existence, length, and form of the input. If there is no possibility of empty output, we may include logics that test to see if the output has value assigned before it is stored.

v. Interface Faults:
    1. In order to make the system work as designed, we need to ensure that each method is correctly invoked. To do so, each developer is responsible to write a short description for what the method does and what each parameter is supposed to be. This allows the user of the method to correctly invoke the method and avoids error.
    2. The database is a shared memory output source. We will build the database such that it is easily accessible by all methods so that it avoids having errors due to type mismatching in between multiple methods.

vi. Storage Management Faults:
    1. When link structure is modified, it is a possible case that the methods that were linked previously are no longer linked. We ensure that all the methods are correctly linked by component/system test and coverage testing. This allows us to see which methods are not used and eases us to find the disconnected links.
    2. As we design the system with the database, we will use dynamic memory data structures. Since we use Python as a main system, it automatically resizes the memory allocation when it gets bigger. However, when it gets smaller, Python does not automatically shrink the data allocation. Thus, although it is a rare case that the dataset that used to have large data is no longer needed to reserve the memory location for upcoming data, we need to pay attention not to exhaust the large size of memory location by leaving them blank.

c. DYNAMIC TESTING: Our goal is to plan for easy implementation of dynamic testing procedures by writing our code with testing in mind. This way we hope to detect bugs early and effectively, to limit time and stress later on in the project lifecycle. We aim to have testing procedures that are done alongside our code, to allow for simple and quick regression testing that can be completed at any time. The specifics of these are outlined in the proceeding bullet point(s) below. Given the timeline of this project, we are not sure if we will get to the point where we will be implementing performance and user testing in its entirety. However, should time allow, we would like to run performance testing of our program to see if it can sustain a high volume of users and keep up with multiple purchases occurring simultaneously. In terms of user testing, we likely won't implement true user testing for our project, given the timeline of the course project and the project objective being more geared towards our learning of the software design principles. We will however implement both validation testing and defect testing throughout the creational lifecycle of our product which we feel will be a suitable surrogate for user testing within our project/ course expectations. We aim to do this each time the team indicates a suitable portion of a component has been created. With our current plan, structured validation and defect testing should

occur roughly every week within each component and interacting components of the system. This will give us a semblance of product verification as well.

d. Specific dynamic testing procedures we are hoping to implement (these are created with reference to the COSC 310 content of Lecture 1 and 2 from Week 5):
    i. Component ("black box") testing:
        1. Completed through validation testing and defect testing of each respective section, as well as the system as a whole, as outlined previously.
    ii. Structural ("white box") testing:
        1. We hope to complete this during our code design where applicable to ensure each portion of the code is working effectively and to also aid in removing redundant (unused) code if present.
        2. We hope to use a combination of the different types of white box testing (linear code sequences, all-definition-use, statement, branch, and path coverage) where applicable. Which is used will be up to the discretion of the individual creating that respective portion of code and with regards to what tests the program optimally.
    iii. Mocking
        1. Given there may be instances where white box testing may not be possible, we will utilize mocking as appropriate. This will be used in situations where we need to test the functionality of a portion of code and its interaction with a given object or component that can be unavailable due to things out of our control. For example, if we are pulling from a server, but the server is down, this has nothing to do with our code, it is something beyond that. Thus mocking is used to truly test the code is operating correctly, and not failing due to occurrences out of our control.

7. Design Patterns

Design Pattern #1 → Behavioural (Observer)

a. With an observer pattern, we have loosely coupled objects that are updated when the object they are "observing" has any state changes. Two good examples of why we need to use this pattern are: in our model there is a host who can edit an event, and any changes to this event will need to be relayed to our users. Another example is anytime a host creates or edits an event, it needs to be approved by an administrator. To implement this, both the host and the admin objects will need to have an observer object that is linked to them. For the host, its observer would be the administrator class that needs to approve any changes or new events the host completes. Then once the administrator approves those

changes, the attendees or individuals monitoring those events will need to be updated of the changes. Thus there will need to be an observer for each event that has all of the observers of that event in it that is notified once the changes are approved by the administrator.

b. See attached UML class diagram and data flow diagrams for representation of the aforementioned relationships.

Design Pattern #2 → Creational (Singleton)

a. A singleton pattern is one which ensures that a class or object has only one instance. This provides a global point to access this object from. This is best for shared resources, which we have within our site. With this in mind, it seems relevant that we would utilize a singleton pattern in our system design. We only want one instance of each event being available to users. A good example of this is if a user is going to purchase a ticket to an event, and if the event object is not up to date and there are no tickets available, we don't want the user going through the entire purchase process, only to find out there actually isn't a ticket available. Thus we will need to make sure there is only one instance of each event to allow everyone access to the most up to date information for that event. To implement this, we will need to use the code discussed in class, where the code double checks it is using only one instance of the object.

Design Pattern #3 → Structural (Facade Pattern)

a. The facade is named so due to its comparison of the exterior of a home (which is called a facade). It is a method by which we hide lower level details and only present the high-level details that are important to the system/ user.
b. We are wanting to keep lower level details hidden and only utilized as needed, as this should help keep our code clean and simple. Additionally it should allow our system to run more efficiently. For example, if the system is creating a new user, it is first going to grab the common information that all users will have (this is the facade). Then depending on the user, it will select the additional items needed for either a host, attendee, or administrative user. In this way, it stream lines the system and allows us to not have to sort through all the different user items, and just select that data that is needed and avoids redundancies or superfluous tasks.
c. Another example could be the use of the systems that we have utilized to create our virtual environment within PyCharm. I want to preface this with I still don't fully understand these systems in their entirety, but this felt like a relevant example as it is low level code that is behind a facade. Within the code we are creating, we are utilizing Flask, PyMongo, Redis, which could be seen as a facade for more complex tasks that these programs perform for us, that we don't see as programmers. So our environment will run our code and then go to these programs as needed to run any items that are deemed necessary by these programs.

d. We felt this would be a good choice because, as discussed above, this will allow our code to maintain simplicity, and ideally make our system more efficient by avoiding redundancies and unneeded tasks.
e. See attached UML class diagram and data flow diagrams for representation of the aforementioned relationships.

Design Pattern #4 → Behavioural (Mediator Pattern)

A. The mediator pattern implements a middle man to manage data flow and plays a role in how objects will interact. We chose this given that our app is one that requires one user that can alter the website (host), which another user is interacting with and/ or is already involved with (attendee). We felt involving a mediator system would help the administrator navigate the task of managing site changes and updates, and also allow for clear communication between hosts and attendees for both upcoming events or events that the user has already purchased. The exact mechanisms of implementing this are still not sorted, but our idea was to have a form of a priority queue that administrative tasks will be put into, so that the administrator can easily navigate and handle these tasks in a timely and effective manner.
B. Another way the mediator pattern could be implemented is to manage updates from the host to the user or user to the host. Likely in a similar fashion to how administrative tasks are held. A rough outline of how this may look is say a host updates an event, and this needs to be communicated to the attendee. We need to make sure we send this to the user via their preferred update method. So this could be handled by a mediator that already has the attendees pre-sorted into groups based on their update preferences (email, sms, etc) and then it takes this information and sends it to the attendees via these preferences. This saves the system from having to go through each individual person who is attending the event and finding their preferred method and sending these one by one.
C. It seems reasonable to note that some of the interactivity above also has overlap with the observer pattern of behavioral design.
D. See attached UML class diagram and data flow diagrams for representation of the aforementioned relationships.